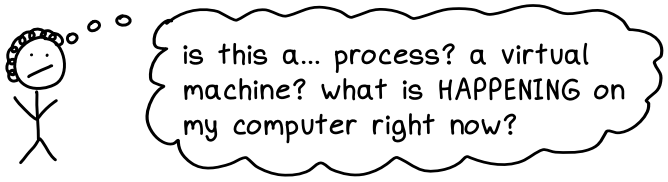
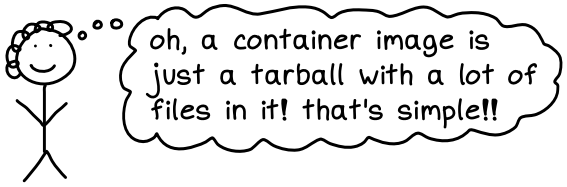
♡ this?
more at
★ wizardzines.com ★

why this zine?

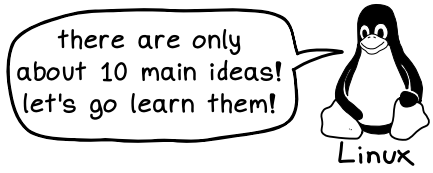
When I started using containers I was SO CONFUSED.



So I decided to learn how they work under the hood!

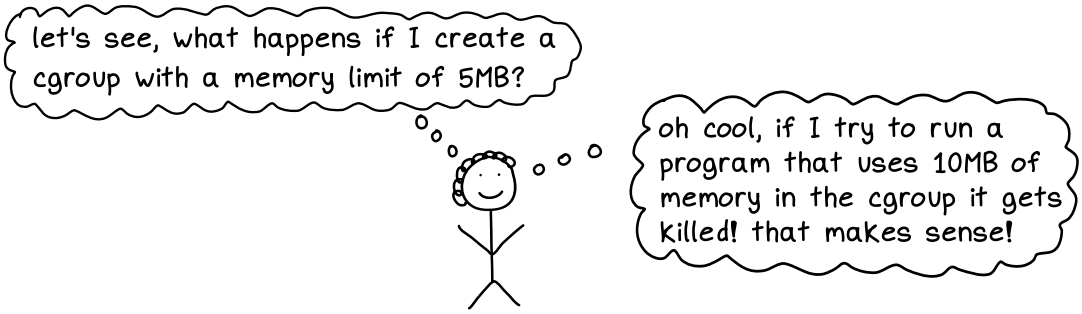


Now I feel confident that I can solve basically any problem with containers because I understand how they work.
I hope that after reading this zine, you'll feel like that too.



♡ thanks for reading ♡

I did a bunch of the research for this zine by reading the man pages.
But, much more importantly, I experimented -- a lot!



So, if you have access to a Linux machine, try things out!
Mount an overlay filesystem! Create a namespace! See what happens!

credits

Cover art: Vladimir Kašiković
Editing: Dolly Lanuza, Kamal Marhubi

configuration options

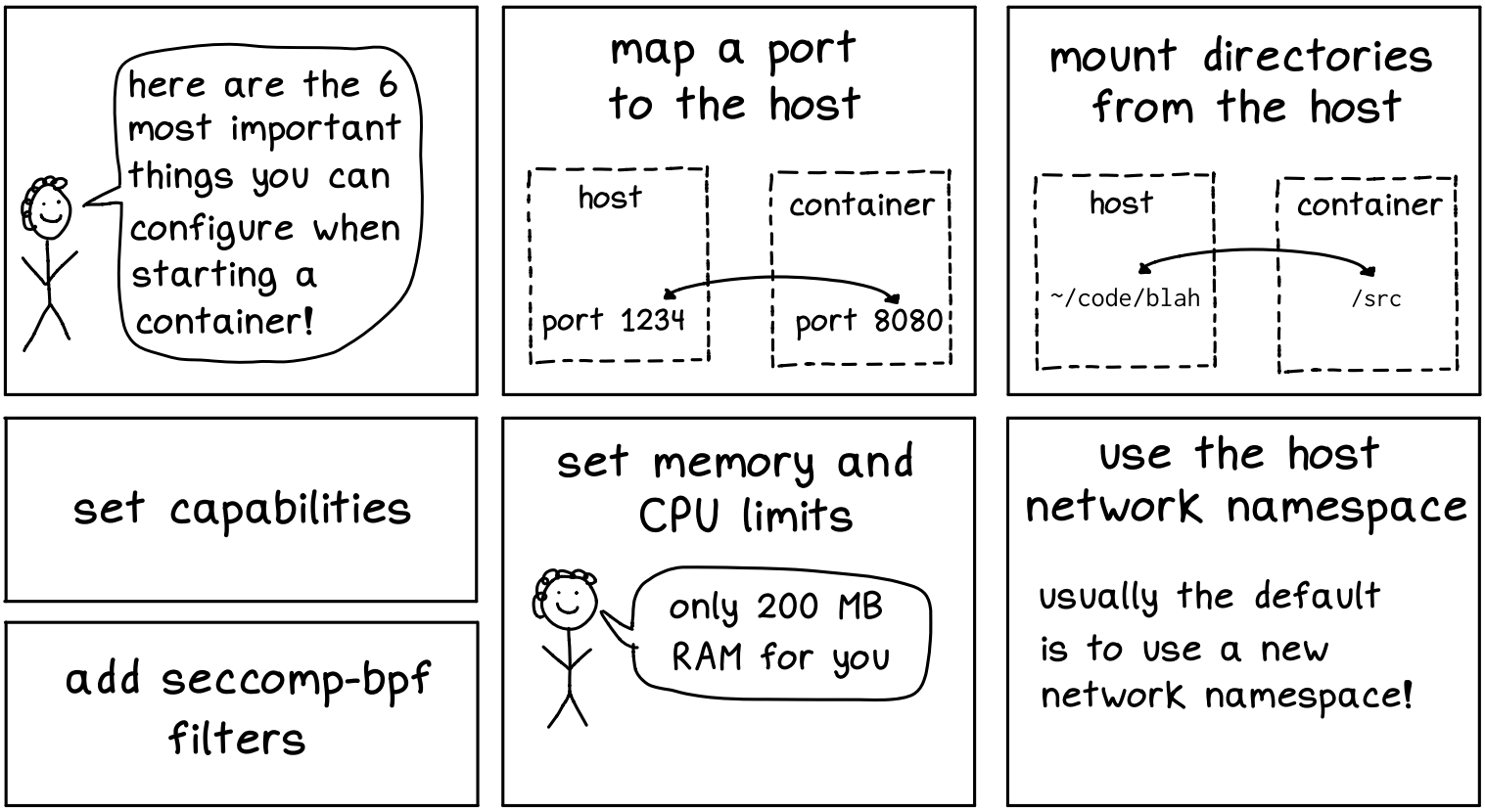


table of contents

why containers?

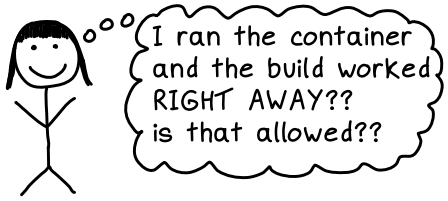
there's a lot of container hype

Here are 2 problems they solve...

problem: building software is annoying

```
$ ./configure
$ make all
ERROR: you have version
       2.1.1 and you need
       at least 2.2.4
```

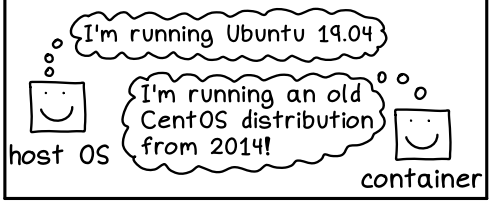
solution: package all dependencies in a ★container★



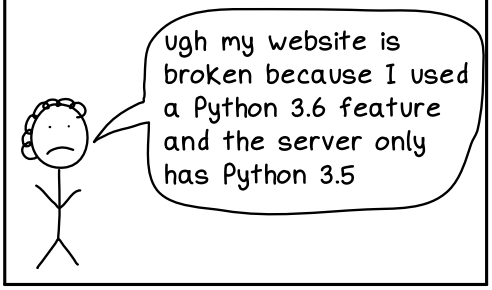
Many CI systems use containers.

containers have their own filesystem

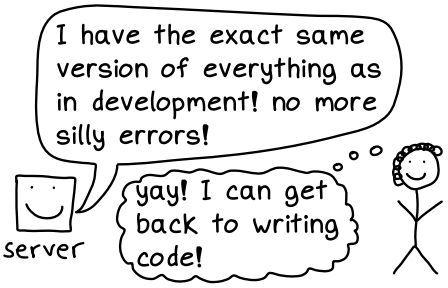
This is the big reason containers are great.



problem: deploying software is annoying too



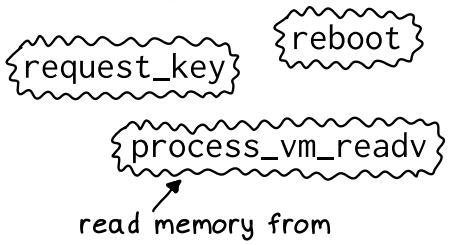
solution: deploy a container



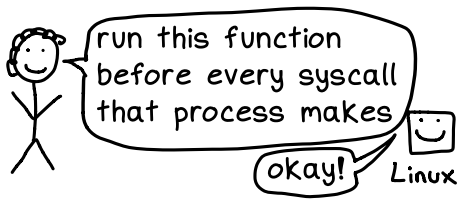
seccomp-bpf

all programs use system calls

rarely-used system calls can help an attacker



seccomp-BPF lets you run a function before every system call



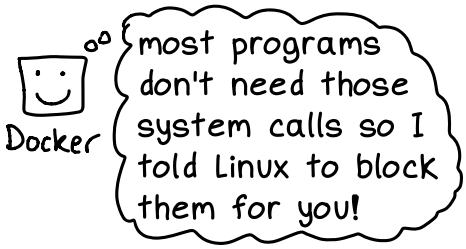
the function decides if that syscall is allowed

example function:

```
if name in allowed_list {
    return true;
}
return false;
```

this means the syscall doesn't happen!

Docker blocks dozens of syscalls by default



2 ways to block scary system calls

1. limit the container's capabilities
2. set a seccomp-bpf whitelist

You should do both!

capabilities

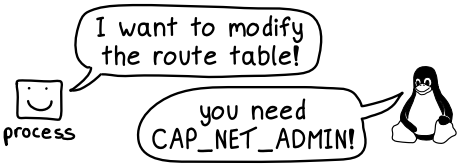
we think of root as being all-powerful...

edit any file

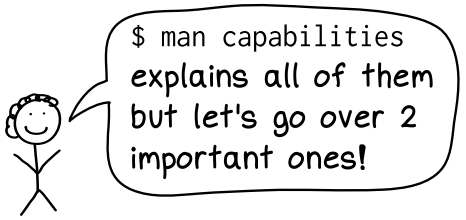
change network config

spy on any program's memory

... but actually to do "root" things, a process needs the right ★capabilities★



there are dozens of capabilities



CAP_SYS_ADMIN

lets you do a LOT of things. avoid giving this if you can!

CAP_NET_ADMIN

allow changing network settings

by default containers have limited capabilities

$ getpcaps PID

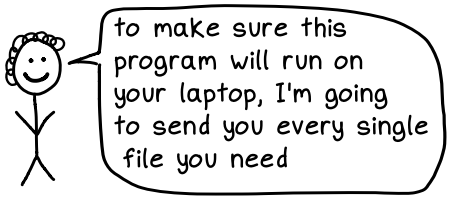
print capabilities that PID has

getcap / setcap

system calls: get and set capabilities!

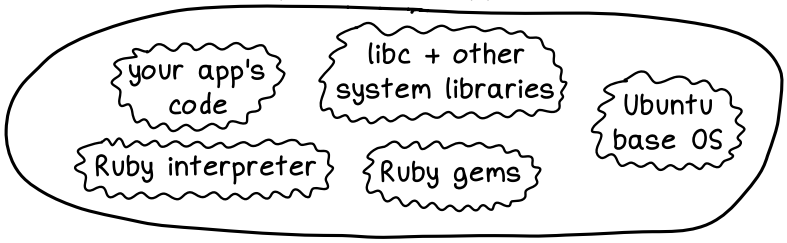
the big idea: include EVERY dependency

containers package EVERY dependency together



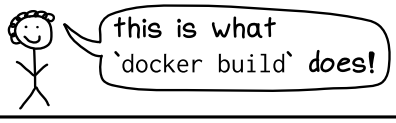
a container image is a tarball of a filesystem

Here's what's in a typical Rails app's container:



how images are built

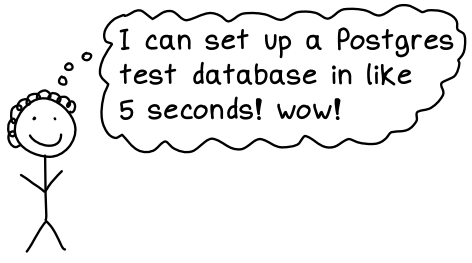
0. start with a base OS
1. install program + dependencies
2. configure it how you want
3. make a tarball of the WHOLE FILESYSTEM



running an image

1. download the tarball
2. unpack it into a directory
3. run a program and pretend that directory is its whole filesystem

images let you "install" programs really easily



containers aren't magic

These 15 lines of bash will start a container running the fish shell. Try it!

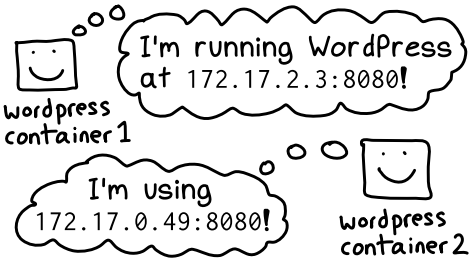
(download this script at bit.ly/containers-arent-magic)

It only runs on Linux because these features are all Linux-only.

```
wget bit.ly/fish-container -O fish.tar       # 1. download the image
mkdir container-root; cd container-root
tar -xf ../fish.tar                          # 2. unpack image into a directory
cgroup_id="cgroup_$(shuf -i 1000-2000 -n 1)" # 3. generate random cgroup name
cgcreate -g "cpu,cpuacct,memory:$cgroup_id"  # 4. make a cgroup &
cgset -r cpu.shares=512 "$cgroup_id"             set CPU/memory limits
cgset -r memory.limit_in_bytes=1000000000 \  #
      "$cgroup_id"                           #
cgexec -g "cpu,cpuacct,memory:$cgroup_id" \  # 5. use the cgroup
    unshare -fmuipn --mount-proc \           # 6. make + use some namespaces
    chroot "$PWD" \                          # 7. change root directory
    /bin/sh -c "                             #
        /bin/mount -t proc proc /proc &&     # 8. use the right /proc
        hostname container-fun-times &&      # 9. change the hostname
        /usr/bin/fish"                       # 10. finally, start fish!
```

container IP addresses

containers often get their own IP address



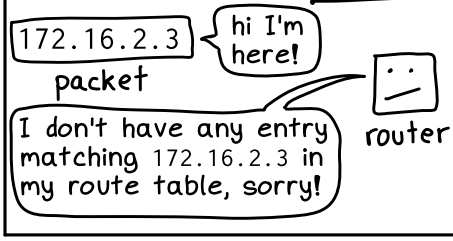
containers use private IP addresses

192.168.*.*
10.*.*.*
172.16.*.* -> 172.32.*.*

} reserved for private networks (RFC 1918)

This is because they're not directly on the public internet

for a packet to get to the right place, it needs a route



inside the same computer, you'll have the right routes

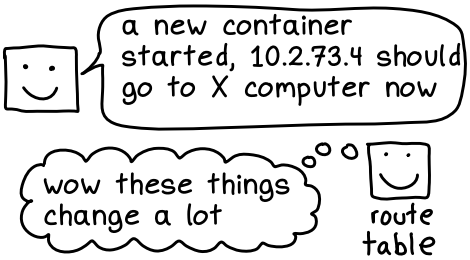
same computer:

```
$ curl 172.16.2.3:8080
<html>....
```

different computer:

```
$ curl 172.16.2.3:8080
.... no reply ....
```

distributing the right routes is complicated

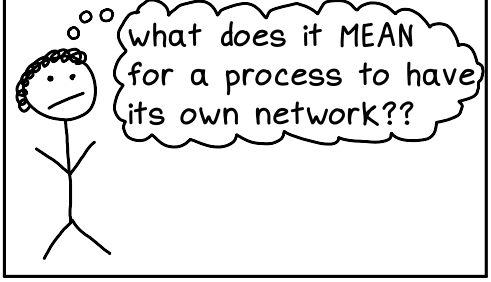


cloud providers have systems to make container IPs work

In AWS this is called an "elastic network interface"

network namespaces

network namespaces are kinda confusing



namespaces usually have 2 interfaces

(+ sometimes more)

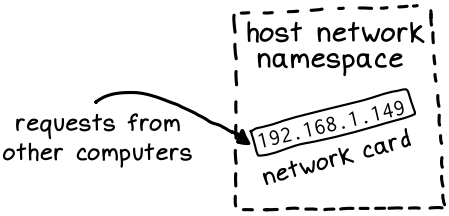
- the loopback interface (127.0.0.1/8, for connections inside the namespace)
- another interface (for connections from outside)

every server listens on a port and network interface(s)

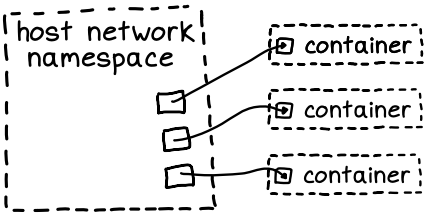
0.0.0.0:8080 means "port 8080 on every network interface in my namespace"

127.0.0.1 stays inside your namespace

your physical network card is in the host network namespace

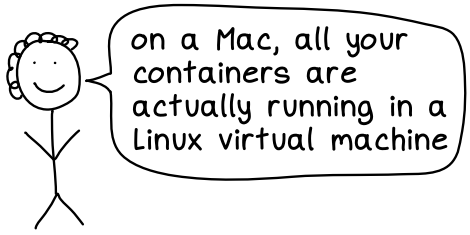


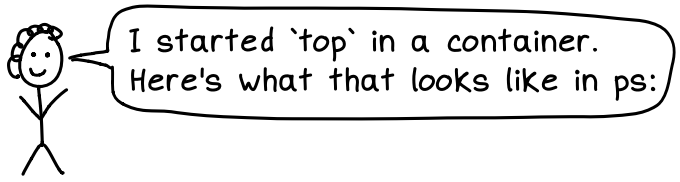
other namespaces are connected to the host namespace with a bridge



containers = processes

a container is a group of Linux processes





outside the container

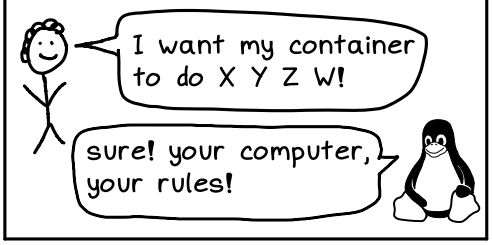
```
$ ps aux | grep top
USER PID START COMMAND
root 23540 20:55 top
bork 23546 20:57 top
```

inside the container

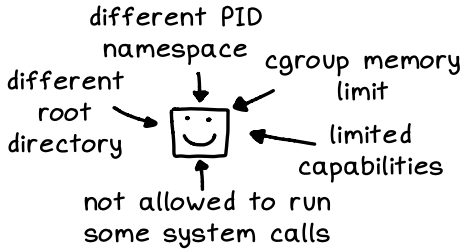
```
$ ps aux | grep top
USER PID START COMMAND
root 25 20:55 top
```

these two are the same process!

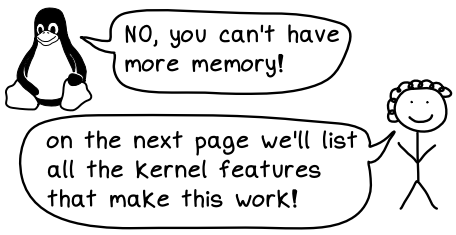
container processes can do anything a normal process can ...



... but usually they have 🔒 restrictions 🔒



the restrictions are enforced by the Linux kernel



container kernel features

containers use these Linux kernel features

"container" doesn't have a clear definition, but Docker containers use all of these features.

♥ pivot_root ♥

set a process's root directory to a directory with the contents of the the container image

★ cgroups ★

limit memory/CPU usage for a group of processes

♥ namespaces ♥

allow processes to have their own:

- → network
- → mounts
- → PIDs
- → users
- → hostname
- + more

★ capabilities ★

security: give specific permissions

♥ seccomp-bpf ♥

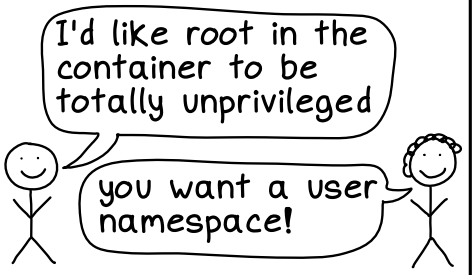
security: prevent dangerous system calls

★ overlay filesystems ★

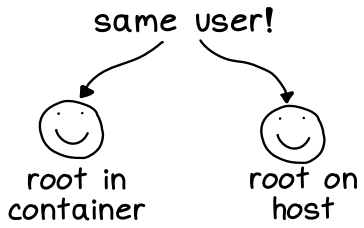
this is what makes layers work! Sharing layers saves disk space & helps containers start faster

user namespaces

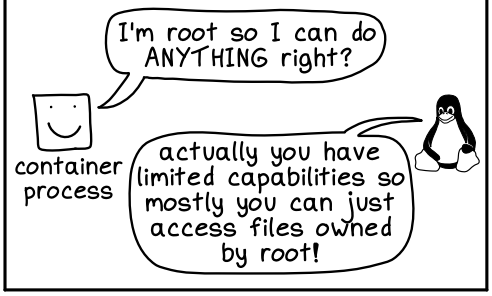
user namespaces are a security feature...



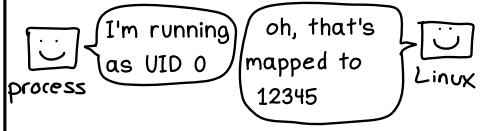
... but not all container runtimes use them



"root" doesn't always have admin access



in a user namespace, UIDs are mapped to host UIDs



The mapping is in `/proc/self/uid_map`

unmapped users show up as "nobody"

create user namespace

```
$ unshare --user bash
$ ls -l /usr/bin
.. nobody nogroup    apropos
.. nobody nogroup    apt
```

these are "actually" owned by root but we didn't map any users

how to find out if you have a separate user namespace

compare the results of

```
$ ls /proc/PID/ns
```

between a container process and a host process.

PID namespaces

the same process has different PIDs in different namespaces

PID in host	PID in container
23512	1
23513	4
23518	12

PID 1 is special

PID namespaces are in a tree

host PID namespace (the root)

child child child

Often the tree is just 1 level deep (every child is a container)

you can see processes in child PID namespaces

if PID 1 exits, everyone gets killed

Killing PID 1 accidentally would be bad

rules for signaling PID 1

from same container:

only works if the process has set a signal handler

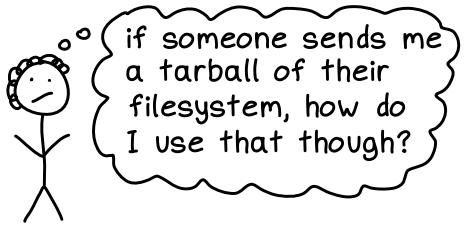
from the host:

only SIGKILL and SIGSTOP are ok, or if there's a signal handler

pivot_root

a container image is a tarball of a filesystem

(or several tarballs: 1 per layer)



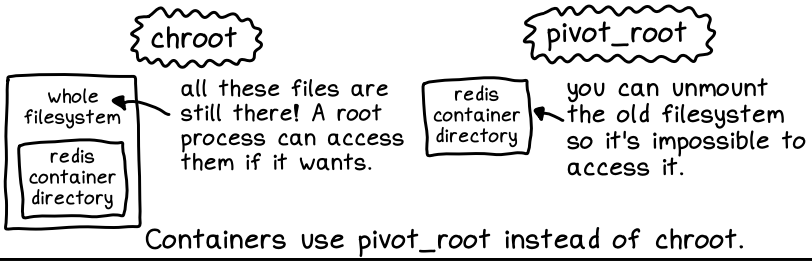
chroot: change a process's root directory

If you chroot to /fake/root, when it opens the file /usr/bin/redis it'll get /fake/root/usr/bin/redis instead.

You can "run" a container just by using chroot, like this:

```
$ mkdir redis; cd redis
$ tar -xzf redis.tar
$ chroot $PWD /usr/bin/redis
# done! redis is running!
```

programs can break out of a chroot



Containers use pivot_root instead of chroot.

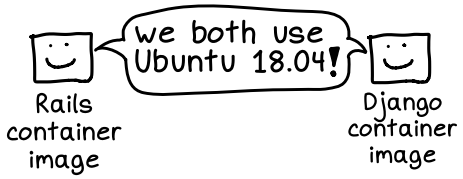
to have a "container" you need more than pivot_root

pivot_root alone won't let you:

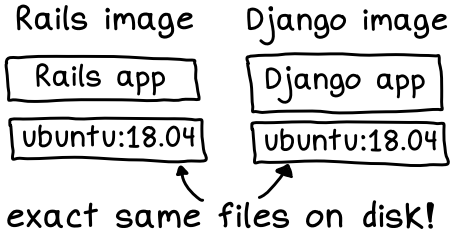
- set CPU/memory limits
- hide other running processes
- use the same port as another process
- restrict dangerous system calls

layers

different images have similar files



reusing layers saves disk space



a layer is a directory

```
$ ls 8891378eb*
bin/ home/ mnt/ run/ tmp/
boot/ lib/ opt sbin/ usr/
dev/ lib64/ proc/ srv/ var/
etc/ media/ root/ sys/
```

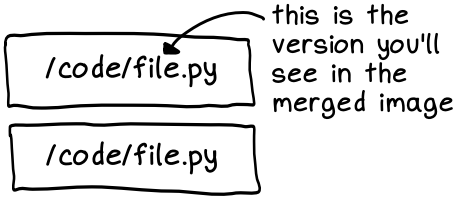
files in an ubuntu:18.04 layer

every layer has an ID

usually the ID is a sha256 hash of the layer's contents

example: `8e99fae2..`

if a file is in 2 layers, you'll see the version from the top layer



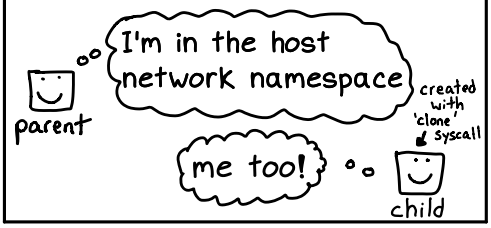
by default, writes go to a temporary layer

these files might be deleted after the container exits

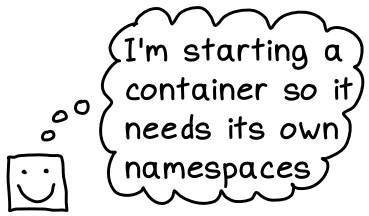
To keep your changes, write to a directory that's mounted from outside the container

how to make a namespace

processes use their parent's namespaces by default



but you can switch namespaces at any time



command line tools

`$ unshare --net COMMAND`

run COMMAND in a new network namespace

`$ sudo lsns`

list all namespaces

`$ nsenter -t PID --all COMMAND`

run COMMAND in the same namespaces as PID

namespace system calls

★ clone ★

make a new process

★ unshare ★

make + use a namespace

★ setns ★

use an existing namespace

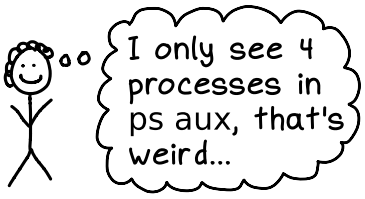
★clone★ lets you create new namespaces for a child process

each namespace type has a ♥ man page ♥

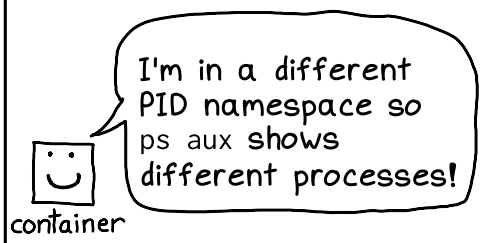
```
$ man network_namespaces
...
A  physical network device
can live in exactly one
network namespace.
...
```

namespaces

inside a container, things look different



why things look different: namespaces



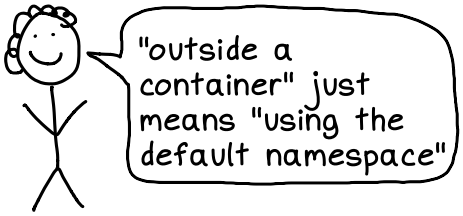
every processes has 7 namespaces

```
          PID
$ lsns -p 273
         NS TYPE
4026531835   cgroup
4026531836   pid
4026531837   user
4026531838   uts
4026531839   ipc
4026531840   mnt
4026532009   net
   namespace ID
```

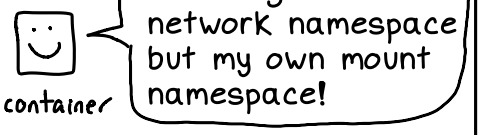
you can also see a process's namespace with:

```
$ ls -l /proc/273/ns
```

there's a default ("host" namespace)

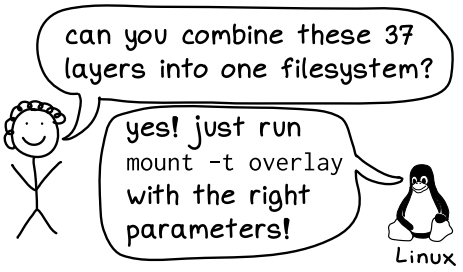


processes can have any combination of namespaces



overlay filesystems

how layers work: mount -t overlay



mount -t overlay has 4 parameters

`lowerdir:`
list of read-only directories

`upperdir:`
directory where writes should go

`workdir:`
empty directory for internal use

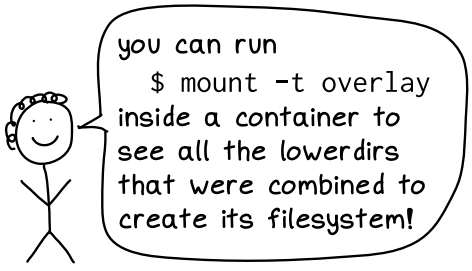
`target:`
the merged result

upperdir: where all writes go

when you create, change, or delete a file, it's recorded in the upperdir.

usually this starts out empty and is deleted when the container exits

lowerdir: the layers. read only.

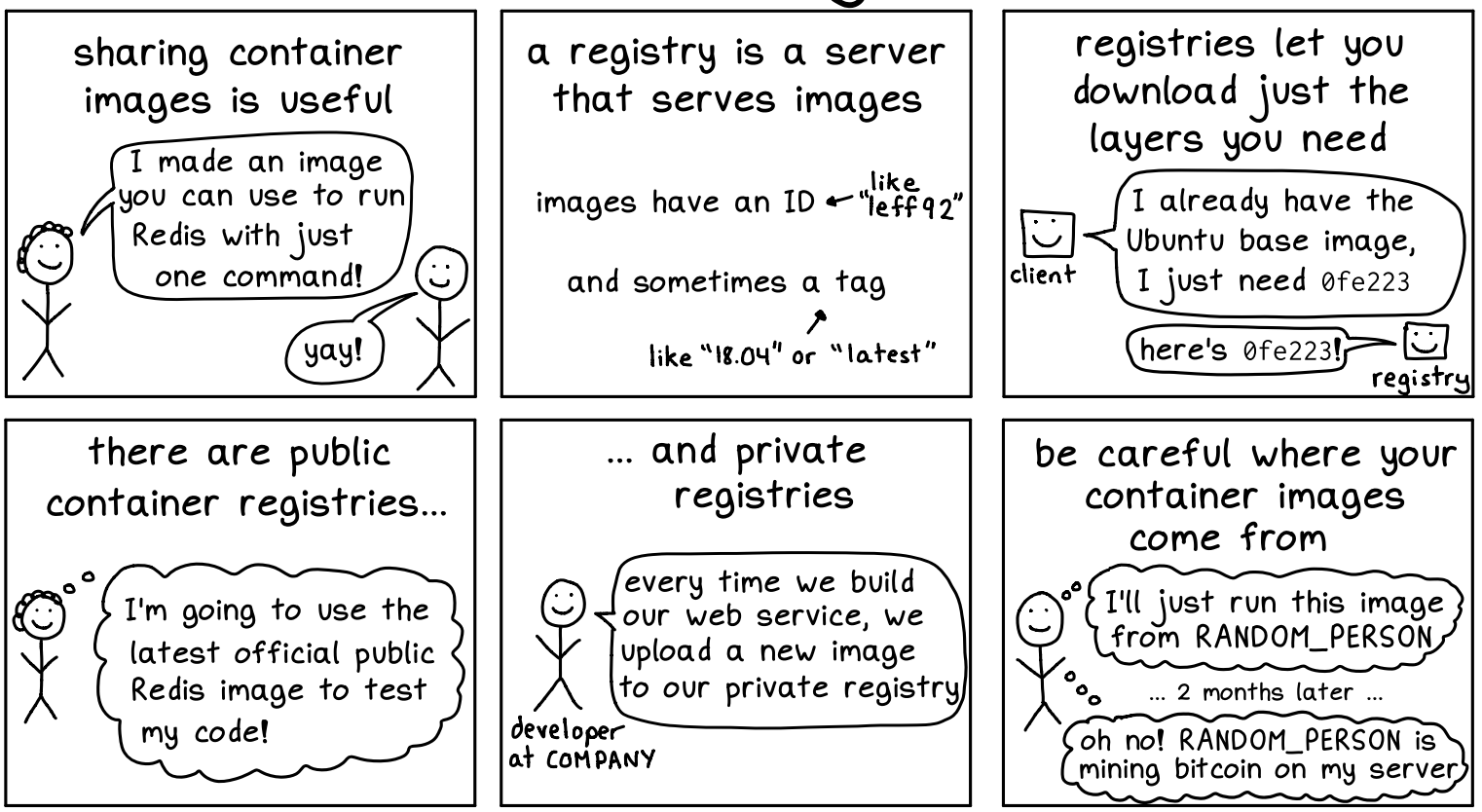


here's an example!

```
$ mount -t overlay overlay -o
    lowerdir=/lower,upperdir=/upper,workdir=/work /merged
$ ls /upper
cat.txt  dog.txt
$ ls /lower
dog.txt  bird.txt
$ ls /merged
cat.txt  dog.txt   bird.txt
```

the merged version of dog.txt is the one from the upper directory

container registries



cgroups